ETH zürich

ETH Library

The remote computation system

Report

Author(s):
Arbenz, Peter; Gander, Walter; Oettli, Michael

Publication date:
1996-04

Permanent link:
https://doi.org/10.3929/ethz-a-006651600

Rights / license:
In Copyright - Non-Commercial Use Permitted

Originally published in:
Technische Berichte 245

This page was generated automatically upon download from the ETH Zurich Research Collection.
For more information, please consult the Terms of use.

The Remote Computation System

P. Arbenz, W. Gander and M. Oettli
Institut für Wissenschaftliches Rechnen,
ETH Zürich, 8092 Zürich

April 24, 1996

Abstract

Today many high performance computers are reachable over some network. However, the access and use of these computers is often complicated. This prevents many users to work on such machines. The goal of our Remote Computation System (RCS) is to alleviate the usage of modern algorithms on high performance computers. RCS has an easy-to-use mechanism for using computational resources remotely. The computational resources available are used as efficiently as possible in order to minimize the response time. We report on experiments involving computations from high-end workstations up to supercomputers.

1 Introduction

Wide area computer networks have become a basic part of today's computing infrastructure. These networks connect a variety of machines, from workstations to supercomputers, presenting an enormous computing resource.

However, the access and the use of these computers and the software is often complicated. A major problem for the inexperienced user to exploit such high performance computers is that he has to deal with machine dependent low level details. The goal of this work is to make high performance computing accessible to scientists and engineers without the need for extensive training in parallel computing and allowing them to use resources best suited for a particular phase of the computation.

This goal shall be achieved with a client-server system, which provides uniform access to modern parallel algorithms on supercomputers. The design and a prototype implementation of such a system, called the *remote computation system* (RCS) is presented here. The user's view of RCS is that of an ordinary software library. The user calls RCS library routines (e.g. to solve a system of linear equations) within his program running on a workstation. In contrast to common libraries, the problem is not necessarily solved on the local workstation, but is dynamically allocated to an arbitrary machine in a given computer cluster, in order to minimize the response time. Further, the system allows distributed applications with several solvers running concurrently on different machines. Furthermore, the RCS routines are generic in the sense that the method with which the problem is solved, is not predetermined. The system tries to solve the problem by the best possible algorithm according to the input arguments. A few additional parameters

would allow a knowledgeable user to precisely steer the problem-solving process while the inexperienced user gets the correct answer in a possibly non-optimal way.

2 System Design

The Remote Computation System has a client-server architecture and consists of two components: the user interface and the run time system. The underlying computational software can be any existing scientific library. This section describes the user interface as well as the internals of a prototype RCS which is restricted to problems from Numerical Linear Algebra.

2.1 User Interface

The interface to the RCS is given in terms of a library of interface routines, which can be called in the user's application. The user writes his application in a platform independent way in Fortran or C as usual. The RCS interface library provides a pair of procedures for each problem, a request and a claim procedure to allow *asynchronous* calls. With the first procedure a request is posted to RCS, i.e., the necessary arguments are passed to RCS. In the example of a linear system, the matrix and the right hand side vector would have to be given. The RCS then decides on the basis of a certain knowledge base and additional parameters given by the user, which algorithm and which host computer, sequential or parallel, is most appropriate to solve the problem. The solver is then started on this machine. In the meantime, the user application continues execution until the claim procedure is called. This procedure blocks until the result is returned from the solver. Thus, the system allows distributed applications running on several machines and in particular, the concurrent execution of two or more solvers.

The following deals with the functionality available to the user through the interface library. Before using any other call to the RCS, the application must enroll into the RCS. The last call to the RCS is to notify that the application does not need its service anymore. The routines

```
subroutine RCinitiate(handle,info)
subroutine RCterminate(handle)
```

enroll the application into RCS and terminate the service respectively. The routine `RCinitiate()` returns a *handle*, an identifier of the local RCS server which is required in any subsequent call to RCS.

Computational problems are submitted to the RCS by posting a request and passing the necessary arguments. The results are then claimed by calling a second routine. For instance, the routines

```
subroutine RCpostSys(n,A,ld,b,host,handle,rqst)
subroutine RCclaimSys(n,x,rqst,info)
```

post the request to solve a linear system of equations $A\mathbf{x} = \mathbf{b}$ and claim the solution $\mathbf{x}$ respectively. The parameter `rqst` which is returned by `RCpostSys()` is required by

`RCclaimSys()` to identify the corresponding result as there might be more than one outstanding result. The argument `host` can be either a valid name of a host in the RCS computer pool or `'*'` if the selection of the host shall be left to the RCS. While `RCpostSys()` returns as soon as the request is accepted by the RCS, `RCclaimSys()` is blocking and returns only after the result has been send by RCS.

Although scientific packages usually provide solvers for different data types (real and complex with various precisions), our interface is currently restricted to real double precision arguments.

A couple of auxiliary routines are available to inquire RCS parameters, the status of a host in the pool or how long it takes to solve a specific problem. For instance, the routine

```
subroutine RCinqSys(n,host,handle,time,info)
```

returns the time it takes to solve a linear system of equations on the host `host`.

Another interesting kind of interface would be for interactive mathematical software packages like MATLAB, Maple or Mathematica. These systems provide a very user-friendly working environment. However their performance is often unsatisfactory for large problems because of the limited power of the workstation it is running on. The RCS may be regarded as a powerful enhancement of such packages. The interface to RCS would consist of a couple of procedures which may be called within such a package. For example, the following MATLAB statements generate a random 480-by-120 matrix A and call RCS *synchronously* to compute its singular value decomposition:

```
>> A = rand(480,120);
>> [u,s,v] = rcs_svd(A);
```

Businger et al. [6] present such an enhancement to Mathematica. Casanova and Dongarra [7] investigate other possible interfaces to a computational system like the RCS.

The usage of the RCS system is easy once it is installed correctly, see Sec. 2.4. When a user wants to run a RCS application, he first starts up the RCS run time system from the UNIX prompt and then runs his application as usual. The RCS can serve multiple applications concurrently.

2.2 Run Time System

The design of the RCS run time system is shown in Figure 1. The server residing on the scientist's workstation represents the core of the system. It accepts requests from the application to solve a certain problem and spawns an appropriate solver on a computer in a predefined pool. The interface library is linked to the user's application. It handles all interprocess communication between the application and the RCS. Similarly, the backend is a simple driver program to which the computational solvers are linked. It handles communication between the solver and the server. Finally, the daemon called *monitor* serves to periodically update dynamically changing parameters such as the workload on each computer. These components of the RCS are now discussed in more detail.

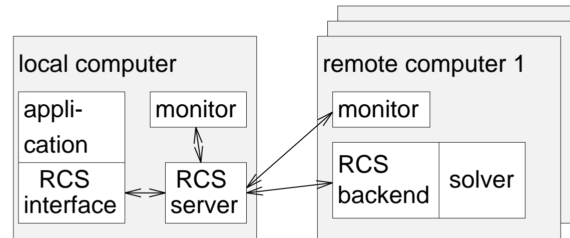

Fig. 1: *components of run time system*

2.2.1 The Server

The server accepts requests from the user's application, determines the most appropriate RCS solver according to the input arguments and starts the solver. If the remote host is not specified by the user, the server selects the solver-host pair such that the response time is minimized. This selection process is described in Section 2.3.

In order to make an optimal choice, the server requires information about

- the request, e.g. problem size, amount of necessary data transfer,
- the problems which RCS can solve,
- the hosts computers in the pool and their characteristics such as number of processors etc.,
- the available computational software (solvers) on each host and their characteristics; for instance, a mathematical model is required to assess its response time,
- dynamically changing parameters as the current workload on each host and the available communication bandwidth on the network.

Most of this information does not change while RCS is running and is read from a configuration file at startup time. Dynamically changing parameters such as the workload of a host computer are periodically measured by monitors residing on each platform.

2.2.2 The Monitor

One criterion influencing the selection of the computational solver is shortest response time. As the UNIX hosts commonly do not run under dedicated conditions, the response time depends on the workload and the network throughput. A simple network monitor is included in RCS which allows to survey the load on the host computers. A daemon called *monitor* resides on each host of the pool to determine this information. (The prototype implementation uses predetermined information about the network throughput only).

In PVM [10], on which the RCS prototype is implemented, it is planned to take the machine load into account when distributing tasks. This feature is currently not yet implemented. At least, the PVM host file permits to specify the raw performance of a host.

Unfortunately, there is no easy way to determine the workload of a host computer. Probably the simplest way is to open a pipe to read from a program that gets this information. Possible programs for getting the machine status are `uptime`, `w`, `who`, `vmstat`, `pstat` or `ps`. They are portable and do not require root permissions to be run. However, this is not proper programming. Another way is to write code that gets the relevant UNIX kernel variables directly. The problems with that approach are that the program may need special permission to run, and it would be less portable, since kernels from different companies are substantially different. A further method to determine the workload is to run a benchmark program and measure cpu time versus wallclock time consumed, e.g. using `getrusage()` and `gettimeofday()`. But this requires to impose a dummy load on the system and is somewhat slower, since the program must run for a while to get a meaningful reading.

The last approach has been chosen for the RCS workload monitor. The daemons, i.e. the monitors, periodically run a short benchmark program. If this workload has changed more than some threshold it is sent to the server. This is because the workload of a host usually stays the same for a longer time and the amount of communication can be kept low.

The above metric of the workload, i.e. the ratio of the cpu time to the wall clock time of a small benchmark program, is suitable for UNIX multiuser, single processor computers, where process scheduling is done by time sharing. The monitor has not been tested on shared-memory multiprocessors however. Distributed-memory multicomputers often do not allow multiple user applications running on a single node. In this case, the workload would simply be determined by the number of nodes in use.

2.2.3 The Computational Software

The underlying computational software is not subject of this work. It is a problem on its own to write reliable and efficient software for sequential or parallel computers. Therefore it is assumed that a computer platform considered to collaborate in the pool, has such software installed. RCS is able to incorporate any package available on the platform provided that an appropriate RCS driver program has been written which calls the specific solver from the package.

The prototype RCS is restricted to problems from numerical linear algebra and uses LAPACK [2] for workstations, vector processors and shared-memory multiprocessors. ScaLAPACK [8] is intended for distributed-memory multicomputers. In addition, in the first part of this project, divide-and-conquer algorithms for solving real symmetric eigenvalue problems and systems of linear equations on multicomputers have been investigated [9, 3, 4].

It has to be noticed however that the concept of the RCS is applicable to a much broader range of applications than those from numerical linear algebra.

2.2.4 Communication Layer

The communication layer could be built on top of the low level TCP/IP protocol. However, tools for implementing distributed programs as the RCS are provided by software packages like PVM [10] or SCIDDLE [5]. These packages allow the utilization of a heterogeneous network of parallel and serial computers as a single computational resource. They provide facilities for spawning and synchronization of and communication between processes over

a network of heterogeneous machines. The advantage of implementing the RCS on top of PVM is portability and ease of use. PVM runs on any UNIX based machine on which the user has a valid account and which is accessible over some network.

2.3 Selecting a Solver

The RCS consists of many computational solvers installed on different hosts in the pool. A specific host may not be able to solve every problem provided by the RCS. But there may be more than one solver to handle a specific problem. Further, the pool of hosts consists of machines with different characteristics from workstations to supercomputers. When the user does not specify the remote host, it is the task of the RCS server to select a computational solver, which is most appropriate to handle a given problem. Furthermore, the selected solver should minimize the response time. To our knowledge, this sort of selection process has not yet been done in the context of a library.

In a first step, the server determines a set of computational solvers which can handle the problem (i.e. solve a symmetric positive definite system of linear equations) and are suitable in terms of memory requirements. All informations required for this first step are found in the configuration file.

In a second step, the server selects the solver in the set from step one, which minimizes the response time. This selection takes into account

- the problem size,
- the computational complexity of each solver,
- the raw execution rate of the solver on the host it is installed on,
- the workload of each host, and
- the bandwidth from the user's machine to each host in the pool.

The application-dependent parameter problem size is included in the problem request sent by the application to the server. The remaining parameters except the machine workload and network bandwidth are regarded as static in the RCS and are defined in the configuration file. The dynamically changing parameters are periodically measured by the monitor daemons and sent to the RCS server. The response time for each solver is assessed with the help of theoretical models based on the above parameters. The Models are described in Section 3.

2.4 System Configuration

The configuration of the RCS, i.e. the pool of host computers, the tasks it can solve and the corresponding computational solvers each host provides are read in from a configuration file at startup. It can not be modified dynamically once the system runs. The only exception is that hosts which got unreachable are removed from the pool.

Below is a simple configuration file defining one problem (task) by its symbolic name, one host and a few parameters describing the computational solver.

```
CONFIGURATION

TASK = Linsys;

HOST vinci;
  path = $HOME/nfp/prg/system;
  arch = serial;
  network = [2.0e-3,5.0e5];    (* latency [s] and bandwidth [Byte/s] *)

  SOLVER linsolve;
    task = Linsys;
    tau = 6.67E-7;                     (* time for one flop [s] *)
    complexity = [0.66,-0.5,0.833,0];  (* coff. of cubic polynomial *)
  END
END
```

One design objective of the RCS is that it should be easy to extend it. The person who administrates RCS should be able to add new problems, hosts or computational solver without much effort. New hosts are added by including their specification in the configuration file, provided that PVM is already installed. New computational solvers require a driver program which is able to communicate with the server, allocates the necessary work space and calls an appropriate library routine to solve the problem. This driver program could be generated automatically based on some specification of the library routine and its arguments. More effort is required when adding a new problem to the prototype RCS. Unfortunately, the RCS administrator has to include the corresponding interface routine in the source of the RCS interface library. A task-independent generic interface would be more suitable.

2.5 Fault Tolerance

An important issue in any distributed system is fault tolerance. Failures may occur due to a network malfunction or to a host disappearance. Assume the RCS server fails to start a solver on a remote host. The server takes this failure into account and tries to start the second best solver. On the other hand, the blocking interface routine to claim a result periodically checks whether the solver is still alive and returns with an error if necessary.

3 Assessing the Response Time

Typically, several hosts in the pool provide a solver for the same problem. If the user does not specify the host in his request, RCS automatically choses the fastest algorithm-computer pair in order to minimize the response time. In order to select the best pair, models of the expected response time for the different choices are required. More precisely, mathematical expressions are required that specify the response time T as a function of parameters like the problem size n, the number of processors p and other algorithm and hardware characteristics:

$$T = f(n, p, \ldots).$$

Modeling the response time requires some knowledge about the implementation of the RCS system. The client (the user's application) sends a request to the RCS server. The server

selects an appropriate solver and starts it on the corresponding host. Then, the input data is directly sent from the client to the solver. After solving the problem, the result is sent back to the client. Hence, the response time consists of three major components which can be examined independently: the solver startup time, the time for data transmission and the actual computation time. Empirical tests on a workstation cluster show that the startup time makes only a small contribution to the total response time unless the problem size is very small, see Section 4.3. Therefore, we are only concerned with the data transmission time and the computation time.

3.1 Modeling the Execution Time

For applications in numerical linear algebra it is commonly agreed that the floating-point operation (flop) count is a good estimation of work. The execution time in turn is the ratio between this amount of work and the computational speed of the computer.

Two difficulties arise with this approach. First, the amount of work is most often problem dependent. Only for direct solvers can operation counts be given as a function of the problem size. For iterative solvers only a crude estimation of work can be given based on some approximation of the convergence rate, see Section 3.1.4.

The second problem is that the speed, i.e., the work divided by time is not a fixed value. The time for executing a floating-point operation is not only hardware-dependent but also context- and operand-dependent. For instance, level-3 BLAS routines [14] are faster than level-2 or level-1 BLAS routines on modern pipelined computers with a hierarchical memory. Or, multiplications which produce denormalized numbers conforming to the IEEE standard are much slower than those producing normalized results on a Sun SPARCstation 1. Therefore, models for modern computers are typically based on an idealized model of the computer taking hardware details such as vector processing capabilities or memory hierarchies into account [8].

Another simpler approach to run time estimation would be to extrapolate from available measurements. It requires a model for the run time as a function of the problem size. Consider for instance solving a linear system by Gaussian elimination. The estimator function is a cubic polynomial in n, the matrix size:

$$T = b_3 n^3 + b_2 n^2 + b_1 n + b_0.$$

The unknown parameters b_i are determined from measurements by a least squares fit. The danger of this approach is that one or even several measurements serve only to determine performance in a narrow region of what is a large, possibly multidimensional, space. Especially for algorithms on high performance computers, they may be a poor indicator of performance in other situations.

Numerous algorithms have been investigated in the context of the idealized PRAM (parallel random access machine) model [1]. However, this is a theoretical model appropriate only for shared-memory multiprocessors. A DLAM (Distributed Linear Algebra Machine) model was introduced in the ScaLAPACK project [8]. It is used to obtain theoretical performance bounds for algorithms in dense linear algebra running on distributed-memory multicomputers. It allows to estimate the execution time before and after implementation and is useful in scalability analysis. Another approach found in the literature is to use

discrete event models to simulate program execution on a certain computer. Such a simulation is very time-consuming and therefore not applicable for simple run time estimations.

3.1.1 Conventional Computers

A suitable execution time model for a conventional unpipelined computer without hierarchical memory is

$$T = \frac{c(n)}{r}, \tag{1}$$

where $c(n)$ is an expression for the number of floating-point operations dependent on the problem size n and r denotes the computational speed. Parameter r is machine- and algorithm-dependent and is approximated by putting the results of a few test runs into equation (1).

The time to perform one floating-point operation is the inverse of the computational rate and is denoted by τ:

$$\tau = 1/r.$$

For example, the time for one floating-point operation on a Sun SPARCstation 10 is $\tau \approx 9.9 \cdot 10^{-8}$ s. This equals a performance rate r of about 10 Mflop/s.

The application of this model is demonstrated in the following example. The observed and estimated run times were compared for three different algorithms given in Table 1. The procedures together with the flop count for $m \times n$-matrices were taken from LAPACK [2]. Figure 2 gives the execution times depending on the problem sizes on a Sun SPARCstation 10. The dashed lines of the model approximate the observed timings fairly well.

Description	Procedures	flop count
LU factorization	`DGETRF`	$\frac{2}{3}n^3 - \frac{1}{2}n^2 + \frac{5}{6}n$
Tridiagonalization	`DSYTRD`	$\frac{4}{3}n^3 + 3n^2 - \frac{17}{6}n - 19$
QR factorization	`DGEQRF`	$2mn^2 - \frac{2}{3}n^3 + mn + n^2 + \frac{14}{3}n$

Table 1: *examined algorithms and flop count*

3.1.2 High Performance Shared-Memory Computers

Modern high performance computers make use of pipelined instruction execution, vector processing, superscalar processing and often have more than one processor sharing a single global main memory, see [12]. Furthermore, they usually incorporate an increased hierarchy of memory layers to alleviate the difference in speed of memory access and computation rate. This group of computers not only consists of the traditional vector-processors and shared-memory multiprocessors (e.g. Cray Y-MP) but also of high-end workstations (e.g. HP J200). Distributed-memory multicomputers are considered in the next section.

Unfortunately, the performance of numerical computations on these high performance computers heavily depend on the number of memory references per floating-point operation. Thus, the observed performance or execution rate depends on the algorithm and

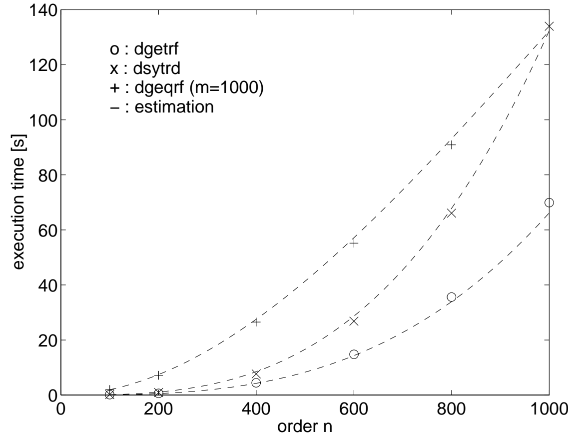

Fig. 2: *observed and estimated run times on a workstation*

problem size. Figure 3 presents the execution rate of the LU factorization depending on the problem size on one processor of a Cray J90 multiprocessor. A valuable model to

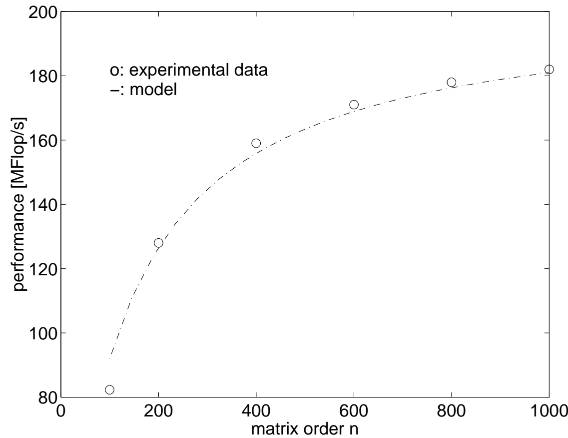

Fig. 3: *performance on one processor of a Cray J90 multiprocessor*

characterize pipelining effects stems from Hockney and Jesshope [13, Sec. 4.5.3]. They model the execution rate $r(n)$ as

$$r(n) = \frac{r_\infty}{\frac{n_{1/2}}{n} + 1}. \tag{2}$$

That is, the performance rate $r(n)$ is expressed in terms of parameters r_∞ and $n_{1/2}$. These parameters are characteristic for a specific algorithm and denote the peak performance for large problem sizes (r_∞) and the problem size for which half the peak performance is reached ($n_{1/2}$).

Again, test runs of the specific algorithm are required to determine these two parameters. An expression $c(n) = c_3n^3 + c_2n^2 + c_1n$ for the number of flops is known from theoretical considerations. Further, it is known that the run time has the form $b_3n^3+b_2n^2+b_1n$. The

parameters b_i may be computed through a least squares fit of the data. As the performance is

$$r(n) = \frac{c_3 n^3 + c_2 n^2 + c_1 n}{b_3 n^3 + b_2 n^2 + b_1 n},$$

the asymptotic performance rate can be predicted by letting n go to infinity. One obtains

$$r_\infty = \frac{c_3}{b_3}.$$

By examining the point at which half the asymptotic rate is reached, $n_{1/2}$ can be determined,

$$\frac{r_\infty}{2} = \frac{c_3}{2b_3} = \frac{c_3 n^3 + \ldots}{b_3 n^3 + b_2 n^2 + \ldots}.$$

Solving for n, one determines that

$$n_{1/2} \approx b_2/b_3.$$

The same model can be used on a shared-memory multiprocessor when parallelism is realized through splitting the loops over the processors. When the number of processors p is increased, r_∞ grows slightly less than linear and $n_{1/2}$ grows faster, because of synchronization effects. Of course, the two parameters have to be determined anew for each number of processors. Consequently, the run time model for an algorithm on a multiprocessor is

$$T = \frac{c(n)}{r(n,p)}, \tag{3}$$

i.e. the flop count divided by the corresponding execution rate as given in equation (2) for a specific number of processor. A run time estimation of the LU factorization on $p = 1, 2, 4, 6, 8, 10$ processors of a Cray J90 is given in Fig. 4. In general, this model gives

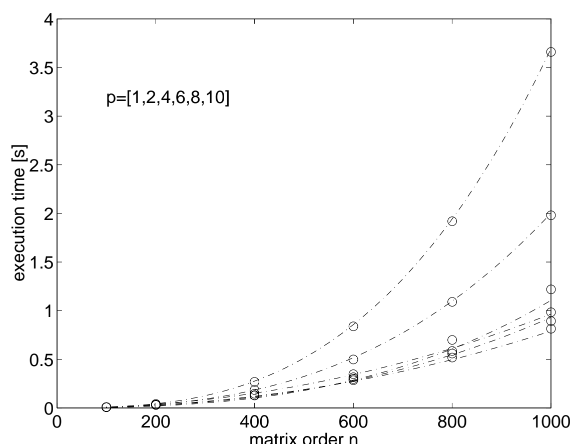

Fig. 4: *run time estimation on a Cray J90 multiprocessor*

a good abstraction of the physical machine. However more hardware details would have to be incorporated to describe possible cache effects, see e.g. [11].

3.1.3 Distributed-Memory Multicomputers

A distributed-memory multicomputer (DMC) consists of multiple processors, often called *nodes*, interconnected by a message-passing network. Each node is an autonomous computer consisting of processor and local memory. The Intel Paragon is an example for a DMC. As the nodes do not share a global memory, processes must interact by passing messages over the interconnection network. Consequently, the execution time of a parallel program on a DMC is the sum of computation time t_c, communication time t_t and idle time t_i on the slowest processor:

$$T = t_c + t_t + t_i.$$

While models to characterize t_c and t_i are considered in the previous two sections, modeling t_t is examined in Section 3.2.

3.1.4 Iterative Algorithms

The work of iterative algorithms does not only depend on the problem size but also on problem-dependent input arguments. Typically, the flop count for one iteration step can be given, but the speed of convergence must be estimated. For instance, the convergence rate of most iterative methods for solving sparse linear systems depends on the extremal eigenvalues of some matrix. Unfortunately, these eigenvalues are commonly not readily available but have to be computed or approximated themselves what is expensive. The situation is even worse for eigenvalue solvers. Their convergence rates directly depend on the spectrum, which is subject to computation. Thus, only rough estimations of work for iterative algorithms can be given in general.

3.2 Data Transfer Time

An idealized model for the time to send a message from one process to another can be based on two parameters: the message startup time t_s, which is the time required to initiate the communication, and the bandwidth b, which tells how many data items can be moved in a certain time interval from the source to the destination process. The time required to send a message of n data items then is

$$T = t_s + \frac{1}{b}n. \tag{4}$$

It typically takes a relatively long time to start up the operation, after which data items arrive at high speed. This idealized model is adequate for many purposes. Nevertheless, it does not take into account the topology of the interconnection network (the Ethernet typically used in LAN's is a bus-based network) and the number of processes communicating at the same time and therefore competing for bandwidth.

Table 2 shows parameters t_s and b for interprocess communication using PVM on different host computers connected by Ethernet. In particular, the computers are a Sun SPARCstation-1 (ru6), a Sun SPARCstation-10 (vinci) and two HP J200 workstations (turing2, turing3) in the institutes local area network as well as a Sun classic (raf2) and a Cray J90 supercomputer (grizzly) in the same building. The values have been determined by a least squares fit of measured communication times in a non-dedicated environment.

Connection	standard routing, XDR encoding t_s [s]	standard routing, XDR encoding b [Byte/s]	direct routing, no encoding t_s [s]	direct routing, no encoding b [Byte/s]
ru6 - vinci	3.6×10^{-3}	3.7×10^{5}	1.4×10^{-3}	8.7×10^{5}
ru6 - turing2	3.3×10^{-3}	3.6×10^{5}	1.5×10^{-3}	7.9×10^{5}
ru6 - raf2	7.3×10^{-3}	2.3×10^{5}	6.1×10^{-3}	6.6×10^{5}
ru6 - grizzly	1.0×10^{-2}	1.4×10^{5}	–	–
turing2 - turing3	1.3×10^{-3}	5.9×10^{5}	5.8×10^{-4}	1.1×10^{6}
turing2 - turing3 †	1.0×10^{-3}	1.0×10^{6}	5.7×10^{-4}	5.7×10^{6}

Table 2: *Communication latency and bandwidth over Ethernet († fast Ethernet)*

A comparison of the two columns make evident that PVM has to be set up properly to get the maximal throughput. By default, PVM uses routing through the PVM daemons and XDR encoding. However, if direct process-to-process links are used and XDR encoding is turned off between computers with the same data format, communication can be sped up considerably. However, XDR encoding, which can make up to 20% of the total message transmission cost in a local ethernet network [16], is inevitable between heterogeneous machines such as a Sun and a Cray J90. The network distance is a further parameter influencing the data transmission throughput.

4 Proof-of-concept

A prototype RCS based on PVM has been implemented to demonstrate its feasibility. PVM (Parallel Virtual Machine) is a software package, which allows the utilization of a heterogeneous network of parallel and serial computers as a single computational resource. The advantage of implementing the RCS on top of PVM is portability and ease of implementation.

This prototype RCS so far provides solvers for linear systems of equations, linear least squares problems and symmetric eigenvalue problems. The underlying computational software is from LINPACK and LAPACK. The RCS itself is a single user system. However, the user may run multiple RCS application concurrently. Prerequisites to running an RCS application is that the user as a valid account on each host in the pool and that RCS has been properly installed by an experienced administrator.

Experiments have been carried in a computer pool consisting of various workstations and a Cray J90 supercomputer. These hosts were connected by a local area Ethernet network.

4.1 An Illustrative Example

Consider to solve a linear system of equations $A\mathbf{x} = \mathbf{b}$, $A \in \mathbb{R}^{n \times n}$ with RCS. An excerpt of the corresponding sample application is given below. The application enrolls in RCS with a call of `RCinitiate()`. Afterward, the problem is posted and its solution claimed using the routines `RCpostsys()` and `RCclaimsys()`, respectively. The following investigations are based on this simple example.

```
program System
  :
  call CreateSystem(n,A,lda,b)
  :
  call RCinitiate(handle,info)
  call RCerror(info)
  :
  call RCpostsys(n,A,lda,b,host,handle,rqsth,info)
  :
  call RCclaimsys(n,x,rqsth,info)
  write 'the solution is:', x
  :
  call RCterminate(handle)
end
```

4.2 Break-even Analysis

Remote computation introduces overhead consisting of the evaluation of the computing request by the server, the starting of the solver as well as the transmission of the necessary data between the local workstation and the remote host. Thus, remote computation only pays off if the problem is sufficiently large and the remote computer is faster or less loaded than the local workstation.

The following experiments shall give an idea at which point remote computation starts to pay off. A generally valid number for the break-even point can not be given as it depends on the specific configuration of a computer pool and the solvers installed. In each experiment, a problem of various size is solved with RCS on a remote host and its execution time is compared with the time of a library routine linked to the user program.

First, a system of linear equations $A\mathbf{x} = \mathbf{b}$ is solved with Gaussian Elimination on a Sun SPARCstation 1 and remotely with RCS on one processor of a HP J200 workstation. Then, the eigenvalues of a symmetric matrix are computed on the same two host computers. The asymptotic computational speed of these computers is 1.5 Mflop/s and 107 Mflop/s respectively. Thus the remote computer is about 70 times faster. The execution times are given in Figure 5 and Figure 6 respectively. The break-even point for the linear system

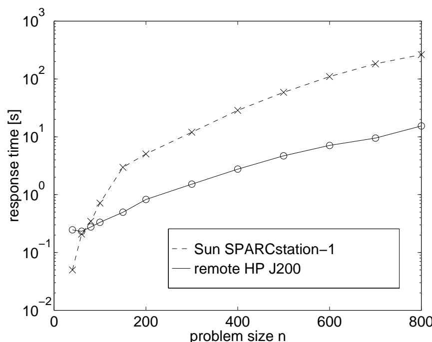

Fig. 5: *linear system*

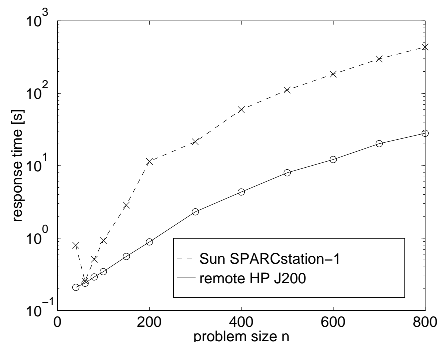

Fig. 6: *symmetric eigenproblem*

is at moderate $n = 100$ and is even lower for the symmetric eigenproblem. It depends on the ratio of remote computation time to the overhead caused by data transfer. As both problems require the same amount of data transfer but the eigenproblem involves more

computation, $\mathcal{O}(\frac{4}{3}n^3)$ compared to $\mathcal{O}(\frac{2}{3}n^3)$ for the linear system, the ratio is better for the eigenproblem. Thus, if more computation is associated with a given data movement, it is relatively less costly.

Of course, the gain in performance is lower if the remote computer is only little faster than the local machine or the network is slow relative to the performance rate of the local workstation. We verify this statement in a second experiment. The linear system is run on two different host configurations, which have about the same ratio of computational speed (the remote host is about ten times faster than the local workstation). However, the absolute performance of the computers in the second configuration is higher. As the interconnection network is the same for both configurations, the relativ cost of data transfer is higher in the second case. Figures 7 and 8 show that the break-even point

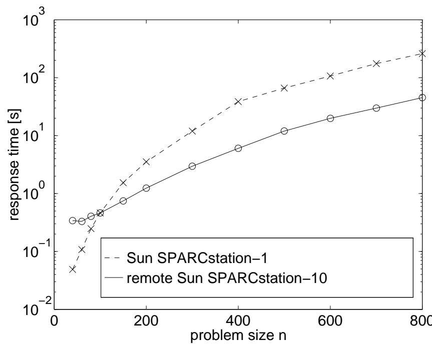

Fig. 7: *low cost data transfer*

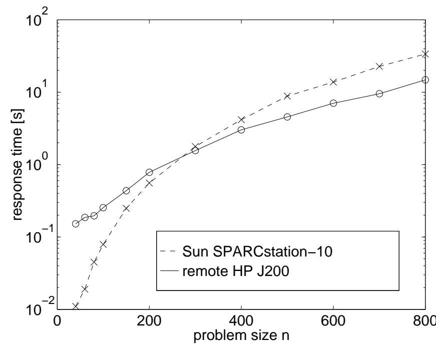

Fig. 8: *high cost data transfer*

is higher and the gain in speed is smaller in the second case due to the higher relative communication overhead.

In the last experiment, RCS runs again on a Sun SPARCstation 1 and the response time of the linear system solver on a Cray J90 is compared with that on the HP J200. Also, the computational power of the Cray J90 is higher than that of the HP workstation, the total response time is higher because of a slower network connection, see Figure 9. Thus, it is not worth using the supercomputer in this situation (unless its large main

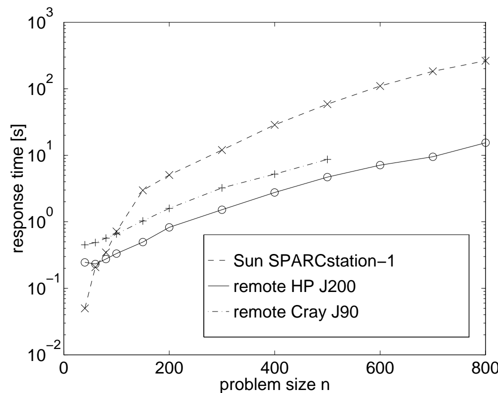

Fig. 9: *remote Cray J90*

memory is required). Such a selection is automatically made by the RCS with the help of

the performance models and the characteristics of the computational resources involved. Section 4.4 shows how good these models match experimental data.

4.3 System Performance

The experiments in the previous section indicate when remote computation helps in terms of speed. Here, we have a closer look at the performance of RCS. That is, the overhead due to data transfer and process start up is examined.

Figure 10 shows that the overhead consisting of start up time and data transfer time makes a major contribution to the overall response time in the experiment presented in Figure 5. The reason is that the remote host (the HP J200) is fast relative to the computer network. Fortunately, the fraction of overhead decreases with n, as the complexity of the computation (a linear system solve) is $\mathcal{O}(n^3)$ whereas the communication goes with $\mathcal{O}(n^2)$.

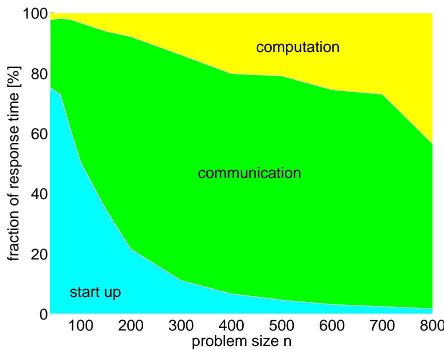

Fig. 10: *remote HP J200*

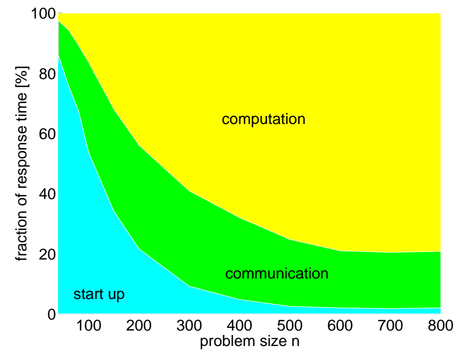

Fig. 11: *remote Sun SPARCstation 10*

The experiment shown in Figure 7 involves much less overhead as can be seen in Figure 11. Nevertheless, the response time in the first situation is shorter as the overhead in both situations is about the same in absolute numbers. Thus, a high percentage of overhead must be taken into account if the response time shall be minimized. In other words, the network-computer configuration is badly balanced in terms of performance.

4.4 Validation of Performance Models

In this section, we compare the theoretical models derived in Section 3 for run time estimations with actual program runs.

The flop count for solving a system of linear equations by Gaussian Elimination is $c(n) = 2/3n^3 - 1/2n^2 + 5/6n$. If the user's workstation is a Sun SPARCstation-1 with an execution rate $r \approx 1.5 \times 10^6$ 1/s (64 Bit precision), then the local execution time T_L is according to equation (1) given by

$$T_L = \frac{2/3n^3 - 1/2n^2 + 5/6n}{1.5 \times 10^6} \quad [s].$$

For the remote computer, an HP J200 workstation, the total response time T_R consists of the computation time T_c, the solver start up time T_s and the data transfer time T_t. The computation time is given by equation (3) with $p = 1$, $r_\infty = 107$ Mflop/s and $n_{1/2} = 393$. The start up time was measured to be $T_s \approx 0.25$ s. Gaussian elimination requires the

transfer of n^2 8-Byte data elements. The data transfer over the Ethernet network is modeled with equation (4) with $t_0 \approx 0.001$ s, $b \approx 5 \times 10^5$ Byte/s. Thus, we get

$$T_R = T_c + T_s + T_t = \frac{2/3n^3 - 1/2n^2 + 5/6n}{\frac{107}{393/n+1}} + 0.25 + \frac{8}{5 \times 10^5} n^2 \quad [s].$$

As the above two models do not include any workload information, they have to be compared to experimental data obtained under dedicated conditions. Figure 12 shows that the model matches the experimental data well.

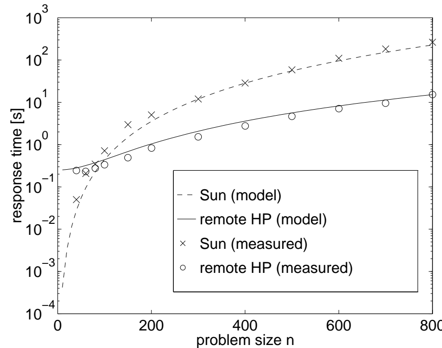

Fig. 12: *Modelled versus measured response times*

4.5 Parallelism

Different types of parallelism can be exploited with the RCS. Because RCS is called asynchronously, it allows distributed applications with several solvers running concurrently on different host computers. Thus, independent subproblems can be solved concurrently. However, the server start up and data transmission is sequential. Therefore, the fraction of computation must be high in order to exploit the parallelism. Only small speedups were obtained with the prototype implementation of RCS.

More promising speedups are obtained by calling a parallel solver on a parallel computer (e.g. LAPACK on multiprocessors or ScaLAPACK on multicomputers). The process to process communication on such computers either through shared memory or a high speed interconnection network is much more efficient than that on a local or even wide area network.

4.6 Discussion

The prototype implementation of the RCS shows the suitability of the design decision. Good estimations of the actual response time have been obtained with the performance models presented in Section 3. Although, the models could be improved by considering further hardware details such as the cache memory, this does not make sense in the context of selecting the fastest of several available solvers in a non-dedicated environment. Other factors such as the proper determination of the workload or the communication bandwidth have more influence on the decision than the omitted hardware details.

Communication is a critical issue. Its performance is low relative to the computation rate of modern workstations and is a major source for further improvements. Surprisingly, it is not the physical data transmission itself but copying data from system in user buffer and data conversion which is expensive (at least in a local area network). These overheads can be reduced in PVM by setting the appropriate parameters. However, things like data conversion are inevitable in a heterogeneous computer pool. The experiments show that local high-end workstations may have a response time comparable to that of supercomputers farther away because of a higher network bandwidth. Then, the only reason for using the supercomputer might be large memory requirements.

An implementational detail which should be mentioned, is that, the interface should be kept more general by providing generic functions to post and claim any problem. This would allow to keep the source code of RCS independent of the problems and tasks it can handle. However, Fortran does not provide the necessary mechanisms to handle variable argument lists.

5 Related Work

Related research is being done at other universities. We mention some projects that have come to our attention.

5.1 The External Computation System

This is a project at the Computer Science Department of the School of Engineering in Biel with the main objective to establish a library of parallel programs in the field of numerical mathematics [6]. The parallel programs are written in a portable way using parallel language extensions to standard C. A simple functional parallel programming system, the External Computation System (ECS), allows to call these programs on a parallel computer within Mathematica running on a workstation.

5.2 The NetSolve Solution Engine

The goal of the NetSolve project [7] at the University of Tennessee at Knoxville is to develop a system which allows users to access computational resources remotely in order to solve scientific problems. Netsolve is designed as fault-tolerant client-server application. Good performance is ensured by a load-balancing policy. On each computer platform in the collection a server process resides which monitors the work load and forks processes that solves the problem using scientific packages. New servers shall be dynamically added to the collection. Ease of use is obtained as a result of different interfaces, some of which do not require any programming effort from the user.

5.3 The Ninf System

Today, many software archives accessible over the Internet provide computational software to the scientific community. This software is typically provided in form of source code, which has to be downloaded by the user to his local computer. There, he has to create

an executable program. Often, it is a tedious task to adapt a piece of software to an inidividual environment. Sometimes, it would be preferable to use computational software by sending the input data to and running the appropriate executable program on some remote host where it is already installed. This is another form of sharing computational resources. However it requires some client-server system which allows to access and run this computational software on a remote host. It is the goal of the Ninf project to develop such system [15].

6 Concluding Remarks

The RCS provides an easy-to-use interface to a variety of numerical linear algebra libraries on UNIX platforms. When the user does not explicitly specify the computer, the RCS is able to select among a number of computer platforms in order to provide the answer in minimal wall clock time.

The RCS is portable and expandable. Its process control and communication is done with PVM assuring that it runs on any UNIX based machine which is accessible over some network. New solvers are easily added to the RCS. Essentially the driver for the new solver has to be provided and a configuration file describing the solvers has to be modified. If the solver provides a new service, corresponding procedures have to be included in the frontend library. So far, such configuration and installation work must be done by a computer scientist.

Often, there is an administrative hindrance in using such an RCS system on expensive high performance computers of a computer center. The RCS system uses remote commands such as a remote shell. These commands are often not available on such computers because of security reasons.

The experimental usage of the prototype system revealed how its extensibility and performance could be further improved. A major drawback of the current implementation is the dependence of the interface library on the problems RCS can solve. When adding a new problem, the interface library has to be modified accordingly. The extensibility of the RCS would be improved by using *generic*, problem-independent interface routines for posting and claiming any problem. A specification of the problems including the number of arguments and its data types could be given in the configuration file. Although, new computational solvers are easily added to the system, its administration could be further simplified by providing a tool to generate driver programs automatically, given some specification of the solvers. Last but not least, the assessement of the response time could be improved by periodically measuring the network throughput as is already done for the workload.

7 Acknowledgments

The authors would like to thank the Swiss National Science Foundation for their support. This project was part of the priority program informatics, module massive parallel computing.

References

[1] S. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice-Hall, 1989.

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide.* SIAM Publications, Philadelphia, 2nd edition, 1994.

[3] P. Arbenz and W. Gander. A survey of direct parallel algorithms for banded linear systems. Technical Report 221, Departement Informatik, ETH Zürich, 1994.

[4] P. Arbenz, W. Gander, and K. Gates. Direct parallel algorithms for banded linear systems. In *Conference on the Priority Programme Informatics Research, Module 3: Massively Parallel Systems, Technopark Zürich*, 1994.

[5] P. Arbenz, H. P. Lüthi, C. Sprenger, and S. Vogel. SCIDDLE: A tool for large scale distributed computing. Technical Report 213, Departement Informatik, ETH Zürich, 1994.

[6] W. Businger, P. A. Chevalier, N. Droux, and W. Hett. Mathematical algorithms for parallel and distributed systems. In *Conference on the Priority Programme Informatics Research, Module 3: Massively Parallel Systems, Technopark Zürich*, 1994.

[7] H. Casanova and J. Dongarra. NetSolve: A network server for solving computational science problems. Technical Report CS-95-313, Department of Computer Science, University of Tennessee at Knoxville, November 1995.

[8] J. Choi, J. Demmel, I Dhillon, J. Dongarra, S. Ostrouchov, A Petitet, K. Stanley, D. Walker, and R. Whaley. LAPACK working note 95: ScaLAPACK: A portable linear algebra library for distributed memory computers – design issues and performance. Technical Report CS-95-283, Department of Computer Science, University of Tennessee, 1995.

[9] K. Gates and P. Arbenz. Parallel divide and conquer algorithms for the symmetric tridiagonal eigenproblem. Technical Report 222, Departement Informatik, ETH Zürich, 1994.

[10] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, Cambridge, 1994.

[11] V. S. Getov. Performance characterisation of the cache memory effect. *Supercomputer*, 11:31–49, 1995.

[12] K. Hwang. *Advanced Computer Architectures.* McGraw-Hill, New York, 1993.

[13] D. Sorensen J. Dongarra, I. Duff and H. van der Vorst. *Solving Linear systems on Vector and Shared Memory Computers.* SIAM Publications, 1991.

[14] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Software*, 5:308–323, 1979.

[15] S. Sekiguchi, M. Sato, H. Nakada, S. Matsuoka, and U. Nagashima. – Ninf –: Network based information library for globally high performance computing. In *Parallel Object-Oriented Methods and Applications (POOMA), Santa Fe*, 1996.

[16] H. Zhou and A. Geist. Receiver makes right data conversion in PVM. In *14th Intel Conference on Computers and Communication, Phoenix*, 1995.